Hey there! How’s it going?

4o mini

**You said:**

Topic - FLIGHT MANAGEMENT SYSTEM Following is the functionality provided by the system: There are two categories of people who would access the system: customer and administrator. Each of these would have some exclusive privileges. 1. The customer can: a. Create his user account. b. Login into the application. c. Check for available flights. d. Make a booking. e. View the bookings made. f. Cancel or modify a booking. 2. The administrator can: a. Login into the application. b. Add flight, schedule and route details. c. View the flight, schedule and route details. d. Cancel or modify the flight, schedule and route details

**ChatGPT said:**

ChatGPT

It sounds like you're designing a **Flight Management System** that handles both customer and administrator functionality. Below is a breakdown of how you might structure the system's features, key interactions, and potential considerations for the customer and administrator roles.

**Flight Management System Overview**

**1. Customer Functionality**

A **Customer** has access to the following features:

**a. Create User Account**

* **Action**: A new user registers by providing necessary personal details like name, email, phone number, and password.
* **Validation**: Ensure unique email, strong password policies (minimum length, mix of characters).
* **Database**: Store customer details securely in the user database.

**b. Login into the Application**

* **Action**: The customer logs in using their registered credentials (email and password).
* **Validation**: Cross-check credentials with the database.
* **Session Management**: Use sessions or tokens (JWT) to manage login state.

**c. Check Available Flights**

* **Action**: The customer can search for flights by specifying departure and destination cities, travel dates, and class preference.
* **Features**:
  + View list of flights based on search criteria.
  + Filter options like price, flight duration, and airline.
* **Database**: Query flight availability and schedules from the flight database.

**d. Make a Booking**

* **Action**: The customer selects a flight from the available list, enters required details (passenger name, contact info, etc.), and proceeds to payment.
* **Payment Integration**: Integration with a payment gateway (e.g., Stripe, PayPal) for completing transactions.
* **Database**: Store booking details in the bookings database (linked to user account).

**e. View Bookings**

* **Action**: The customer can view their past and upcoming bookings with details such as flight number, date, time, and seat class.
* **Database**: Query the customer’s past bookings from the database and display them in an organized format.

**f. Cancel or Modify a Booking**

* **Action**: The customer can cancel or modify existing bookings, subject to airline policies (e.g., refund rules, change fees).
* **Modification Options**: Change flight dates, seats, or passenger details.
* **Database**: Update the booking details in the database.
* **Notifications**: Send confirmation emails or notifications for cancellations or modifications.

**2. Administrator Functionality**

An **Administrator** has a broader range of capabilities for managing the flight system:

**a. Login into the Application**

* **Action**: Admin logs in using their credentials (admin email and password).
* **Validation**: Ensure the admin credentials match the records in the database.

**b. Add Flight, Schedule, and Route Details**

* **Action**: The administrator can add new flights, schedules, and routes to the system.
  + **Flight Details**: Flight number, airline, aircraft, seat availability, etc.
  + **Schedule Details**: Departure time, arrival time, and frequency.
  + **Route Details**: Start and destination airports, intermediate stops, etc.
* **Database**: Insert new flight, schedule, and route data into the respective tables in the database.

**c. View Flight, Schedule, and Route Details**

* **Action**: Admin can view the details of all flights, schedules, and routes in the system.
  + This could include a list view, search filters, and pagination if the system contains a large number of flights.

**d. Cancel or Modify Flight, Schedule, and Route Details**

* **Action**: Admin can modify or cancel existing flights, schedules, or routes.
  + **Flight Modification**: Change flight timings, seat availability, or other details.
  + **Route Modification**: Add or remove intermediate stops, change departure/destination airports.
* **Database**: Update the relevant database tables with new flight or route details.
* **Notifications**: Notify customers of any cancellations or modifications (for example, flight delays, cancellations).

**System Design Considerations**

**1. Security**

* Ensure that customer data is stored securely (e.g., password hashing, encryption for payment info).
* Admin functionalities should be restricted and require a secure, role-based authentication system.

**2. Database Structure**

* **Tables for Users**: Store customer and admin details.
* **Tables for Flights**: Includes flight number, route, schedule, seats, and status.
* **Tables for Bookings**: Links customers with their bookings, including payment and modification history.

**3. User Interface (UI)**

* The system should have an easy-to-navigate interface for both customers and administrators.
  + **Customer Dashboard**: Display flight search, booking status, and account details.
  + **Admin Dashboard**: Display tools for adding/editing flights, schedules, and routes.

**4. Notifications**

* **Customer Notifications**: Send confirmations for bookings, cancellations, and modifications (email, SMS, or in-app notifications).
* **Admin Notifications**: Alerts for any system issues (e.g., overbooked flights) and customer requests.

**5. Flight Availability and Booking Integrity**

* Ensure that the system properly handles concurrent booking requests, preventing overbooking.
* Implement a real-time booking system with a lock on flight availability until the customer completes the payment process.

**6. Performance & Scalability**

* Ensure the system can handle a large volume of bookings, especially if you expect high traffic.
* Consider implementing caching for frequently accessed data (e.g., flight schedules, routes) to improve performance.

**Sample User Flow**

**Customer Journey:**

1. **Account Creation**: A new customer registers and creates a user account.
2. **Login**: The customer logs in to the system using email and password.
3. **Flight Search**: The customer searches for available flights by entering departure and destination cities, and travel dates.
4. **Booking**: After selecting a flight, the customer proceeds to enter details and make a payment.
5. **Booking Confirmation**: The system confirms the booking and sends an email or SMS confirmation.
6. **Modify/Cancel Booking**: Later, the customer might choose to modify or cancel the booking based on availability and policies.

**Administrator Journey:**

1. **Login**: The admin logs into the backend system.
2. **Manage Flights**: The admin adds new flight schedules, routes, and modifies existing ones.
3. **View Bookings**: The admin may want to check the status of bookings or handle customer queries.
4. **Update/Cancel**: If needed, the admin may cancel or modify flight details or notify customers about changes.

This is a high-level breakdown. You can further expand each functionality depending on your system's complexity, adding features like user roles (admin, support staff), payment gateway integration, etc. Let me know if you need any more detailed design or specific help with parts of this system!

4o mini

**You said:**

give the project structure and code for this topic using spring boot, microservices and jwt give full project code

**ChatGPT said:**

ChatGPT

To create a **Flight Management System** using **Spring Boot**, **Microservices**, and **JWT Authentication**, we will structure the project in a modular way. This will include multiple microservices that handle different aspects of the system such as:

1. **User Service** - Handles user management (for both customers and administrators).
2. **Flight Service** - Manages flight data, schedules, and routes.
3. **Booking Service** - Handles flight bookings, cancellations, and modifications.
4. **Authentication Service** - Handles JWT authentication and user login.

We will also implement security using **JWT** to authenticate users and authorize them based on their roles (customer/admin).

Here is a step-by-step breakdown of how to implement this system.

**Project Structure**

The project will consist of the following modules:

* **flight-management-system** (Parent Project)
  + **user-service** (Handles user account creation and authentication)
  + **flight-service** (Handles flight and schedule management)
  + **booking-service** (Handles flight booking, cancellations, and modifications)
  + **auth-service** (Handles JWT authentication)

Each service will be a separate Spring Boot microservice running on different ports.

**Dependencies**

You will need the following dependencies in your pom.xml for each module:

* **Spring Boot Starter Web**
* **Spring Boot Starter Security**
* **Spring Boot Starter Data JPA**
* **Spring Boot Starter Actuator**
* **Spring Boot Starter OAuth2 (for JWT)**
* **Spring Boot Starter Validation**
* **H2 Database (for demo)** or **MySQL** (in production)

Here is an example of the dependencies for each microservice:

xml

Copy code

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.11.5</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<!-- Use H2 for in-memory database for development, or use MySQL -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

</dependencies>

**Service Breakdown**

**1. User Service**

This service will manage user account creation, authentication, and authorization.

**UserController.java**

java

Copy code

package com.flightmanagement.userservice.controller;

import com.flightmanagement.userservice.model.User;

import com.flightmanagement.userservice.service.UserService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api/users")

public class UserController {

@Autowired

private UserService userService;

@PostMapping("/register")

public User registerUser(@RequestBody User user) {

return userService.registerUser(user);

}

@PostMapping("/login")

public String loginUser(@RequestBody User user) {

return userService.loginUser(user);

}

}

**UserService.java**

java

Copy code

package com.flightmanagement.userservice.service;

import com.flightmanagement.userservice.model.User;

import com.flightmanagement.userservice.repository.UserRepository;

import com.flightmanagement.userservice.security.JwtTokenProvider;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.stereotype.Service;

@Service

public class UserService {

@Autowired

private UserRepository userRepository;

@Autowired

private JwtTokenProvider jwtTokenProvider;

private BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

public User registerUser(User user) {

user.setPassword(passwordEncoder.encode(user.getPassword()));

return userRepository.save(user);

}

public String loginUser(User user) {

User existingUser = userRepository.findByUsername(user.getUsername());

if (existingUser != null && passwordEncoder.matches(user.getPassword(), existingUser.getPassword())) {

return jwtTokenProvider.createToken(existingUser.getUsername(), existingUser.getRole());

}

throw new RuntimeException("Invalid credentials");

}

}

**JwtTokenProvider.java**

java

Copy code

package com.flightmanagement.userservice.security;

import io.jsonwebtoken.Jwts;

import io.jsonwebtoken.SignatureAlgorithm;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

import java.util.Date;

@Component

public class JwtTokenProvider {

@Value("${jwt.secret}")

private String secretKey;

@Value("${jwt.expiration}")

private long validityInMilliseconds;

public String createToken(String username, String role) {

Date now = new Date();

Date validity = new Date(now.getTime() + validityInMilliseconds);

return Jwts.builder()

.setSubject(username)

.claim("role", role)

.setIssuedAt(now)

.setExpiration(validity)

.signWith(SignatureAlgorithm.HS256, secretKey)

.compact();

}

}

**User.java (Entity)**

java

Copy code

package com.flightmanagement.userservice.model;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class User {

@Id

private String username;

private String password;

private String role; // Role can be "customer" or "admin"

// Getters and setters

}

**UserRepository.java (Repository)**

java

Copy code

package com.flightmanagement.userservice.repository;

import com.flightmanagement.userservice.model.User;

import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, String> {

User findByUsername(String username);

}

**2. Flight Service**

The **Flight Service** handles flight and schedule data.

**FlightController.java**

java

Copy code

package com.flightmanagement.flightservice.controller;

import com.flightmanagement.flightservice.model.Flight;

import com.flightmanagement.flightservice.service.FlightService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/flights")

public class FlightController {

@Autowired

private FlightService flightService;

@PostMapping

public Flight addFlight(@RequestBody Flight flight) {

return flightService.addFlight(flight);

}

@GetMapping

public List<Flight> getAvailableFlights() {

return flightService.getAvailableFlights();

}

}

**FlightService.java**

java

Copy code

package com.flightmanagement.flightservice.service;

import com.flightmanagement.flightservice.model.Flight;

import com.flightmanagement.flightservice.repository.FlightRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class FlightService {

@Autowired

private FlightRepository flightRepository;

public Flight addFlight(Flight flight) {

return flightRepository.save(flight);

}

public List<Flight> getAvailableFlights() {

return flightRepository.findAll();

}

}

**Flight.java (Entity)**

java

Copy code

package com.flightmanagement.flightservice.model;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class Flight {

@Id

private String flightNumber;

private String airline;

private String departureTime;

private String arrivalTime;

private String from;

private String to;

// Getters and setters

}

**FlightRepository.java (Repository)**

java

Copy code

package com.flightmanagement.flightservice.repository;

import com.flightmanagement.flightservice.model.Flight;

import org.springframework.data.jpa.repository.JpaRepository;

public interface FlightRepository extends JpaRepository<Flight, String> {

}

**3. Booking Service**

This service will manage flight bookings, cancellations, and modifications.

The booking service will need to interact with the **User Service** to validate users and with the **Flight Service** to manage available flights.

**4. Authentication Filter**

To secure the system, we need an **Authentication Filter** to check JWT in every request.

**JwtAuthenticationFilter.java**

java

Copy code

package com.flightmanagement.authservice.security;

import io.jsonwebtoken.Jwts;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import org.springframework.stereotype.Component;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.web.filter.OncePerRequestFilter;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.http.HttpHeaders;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.ServletException;

import java.io.IOException;

import java.util.Collections;

@Component

public class JwtAuthenticationFilter extends OncePerRequestFilter {

private static final String SECRET\_KEY = "yourSecretKeyHere";

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

String token = request.getHeader(HttpHeaders.AUTHORIZATION);

if (token != null && token.startsWith("Bearer ")) {

String jwt = token.substring(7);

String username = Jwts.parser()

.setSigningKey(SECRET\_KEY)

.parseClaimsJws(jwt)

.getBody()

.getSubject();

if (username != null) {

UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(username, null, Collections.emptyList());

SecurityContextHolder.getContext().setAuthentication(authentication);

}

}

filterChain.doFilter(request, response);

}

}

**Running the System**

* Set up the database (H2 or MySQL).
* Create the necessary tables for users, flights, bookings, etc.
* Configure application properties (application.properties) for each service.

To test, you can:

* Register a user via /api/users/register.
* Log in with the user via /api/users/login to get a JWT token.
* Use this token to access secured endpoints in other services (flight booking, viewing flights).

**Conclusion**

This provides the base structure for your **Flight Management System** with Spring Boot, Microservices, and JWT authentication. You will need to extend and adjust this depending on specific business requirements (e.g., booking constraints, payment integration, etc.).

Let me know if you need any further help!

4o mini

Top of Form

Bottom of Form